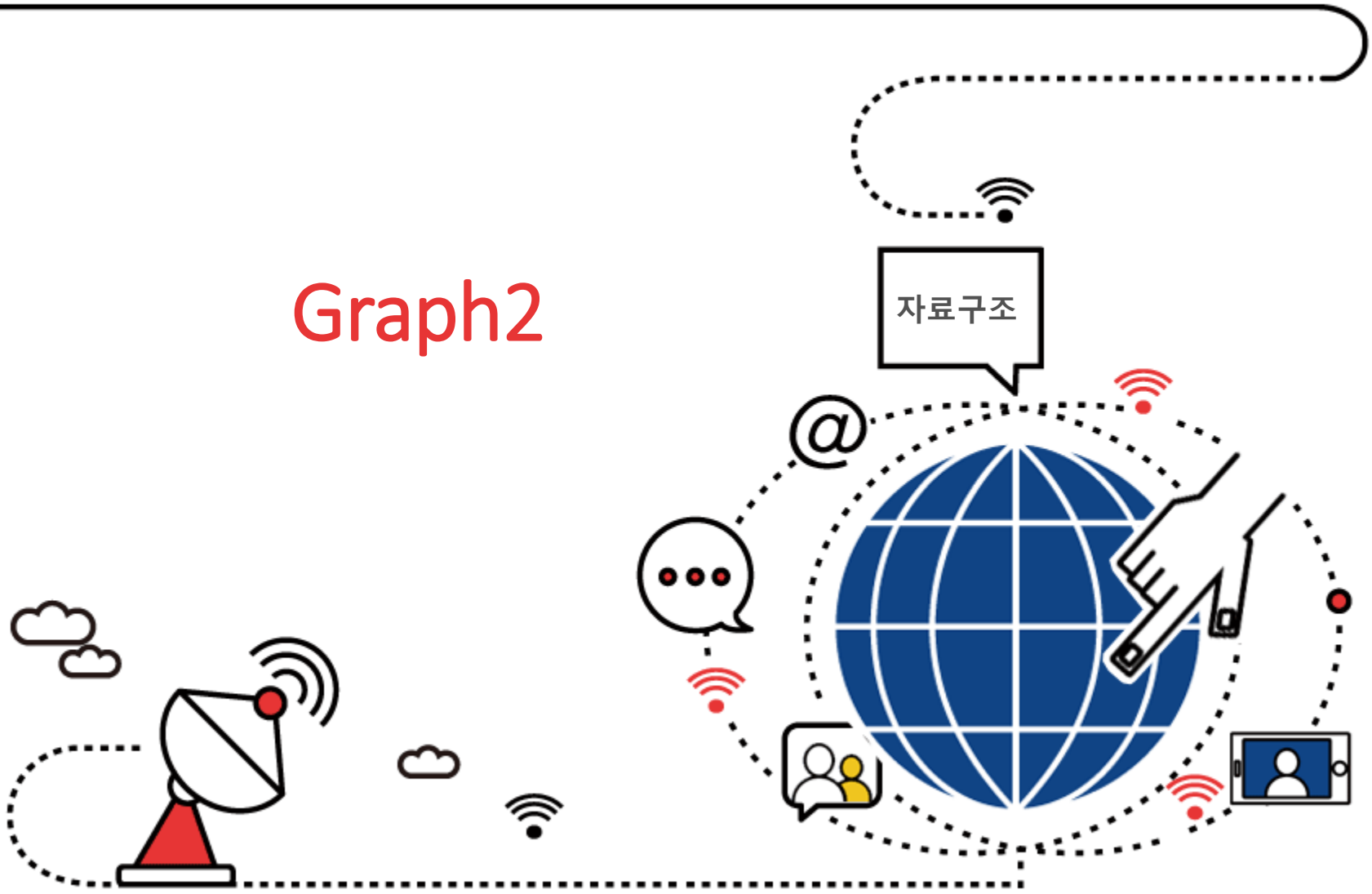
Graph2

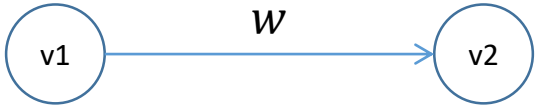


Graph 란?

- 정점(Vertex)과 간선(Edge)으로 이루어진 자료구조
- 간선의 종류에 따라 그래프 종류가 달라진다
  - Undirected Graph : 간선이 방향 X

  - Directed Graph : 간선이 방향을 가리킴

  - Weighted Graph : 간선에 가중치가 있음



Graph

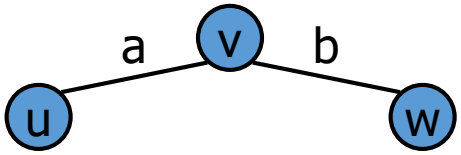
- **Graph** $G = (V, E), n = |V|, m = |E|$
- **Weighted Graph** $G = (V, E, W)$
- **Graph의 표현방법(Representation)은 2가지**
  - **Adjacency Matrix Representation(인접 행렬)**
  - **Adjacency List Representation (인접 리스트)**

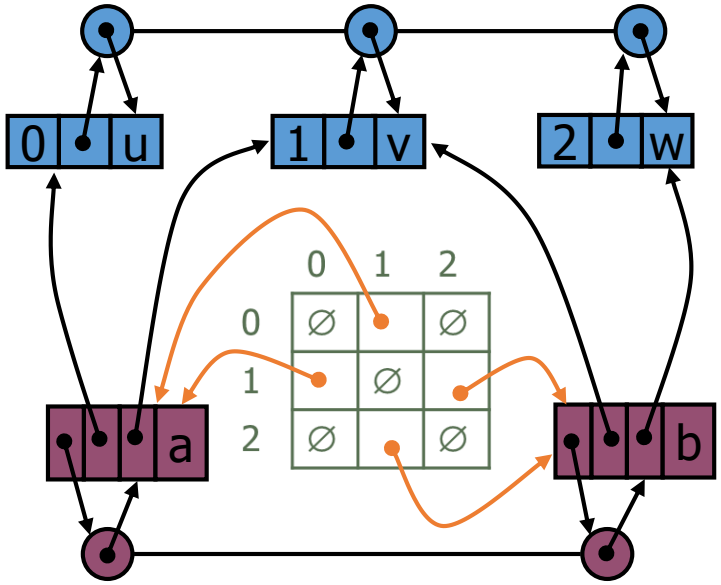
▪ no parallel edges ▪ no self-loops	Adjacency List	Adjacency Matrix
Space	$n + m$	n^2
v.incidentEdges()	deg(*v*)	*n*
v.isAdjacentTo (*w*)	min(deg(*v*), deg(*w*))	1
insertVertex(*o*)	1	n^2
insertEdge(*v*, *w*, *o*)	1	1
eraseVertex(*v*)	deg(*v*)	n^2
eraseEdge(*e*)	1	1

인접행렬 표현법

인접 행렬 표현법 (Adjacency Matrix Representation)

- Graph에 대한 인접 행렬은 $|V| \times |V|$의 행렬을 사용한다
- 노드 i와 j사이에 간선이 존재하면 간선의 pointer, 아니면 NULL
- Weighted Graph일 경우 간선에 weight 값을 저장한다

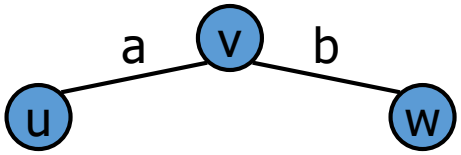


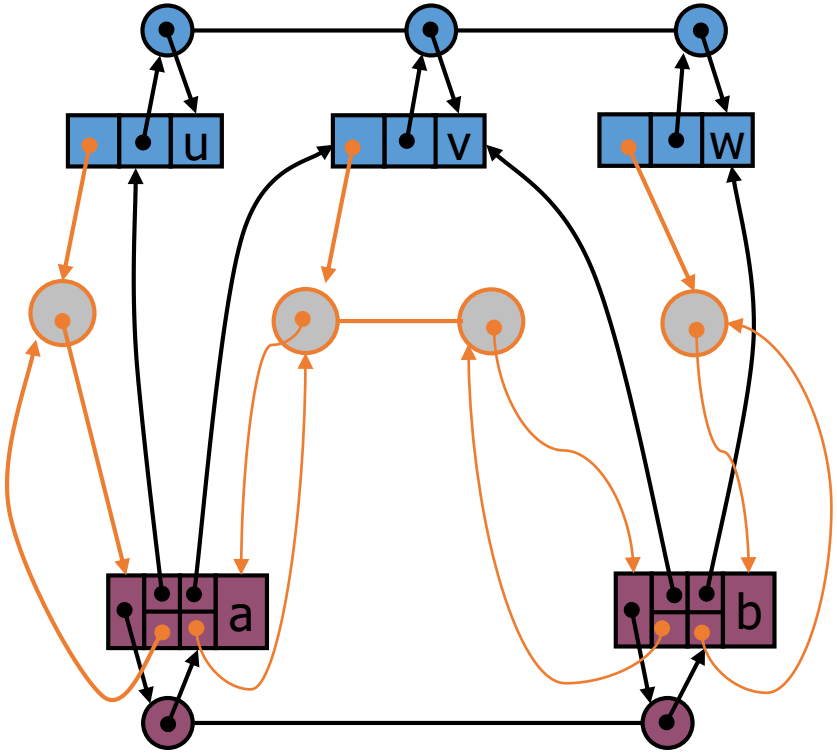


인접리스트 표현법

인접 리스트 표현법 (Adjacency List Representation)

- 임의의 노드 i 에 대하여 인접한(incident) 간선 또는 인접한(adjacent)정점들에 직접적인 접근 지원
- Weighted Graph일 경우 간선에 weight 값을 저장한다

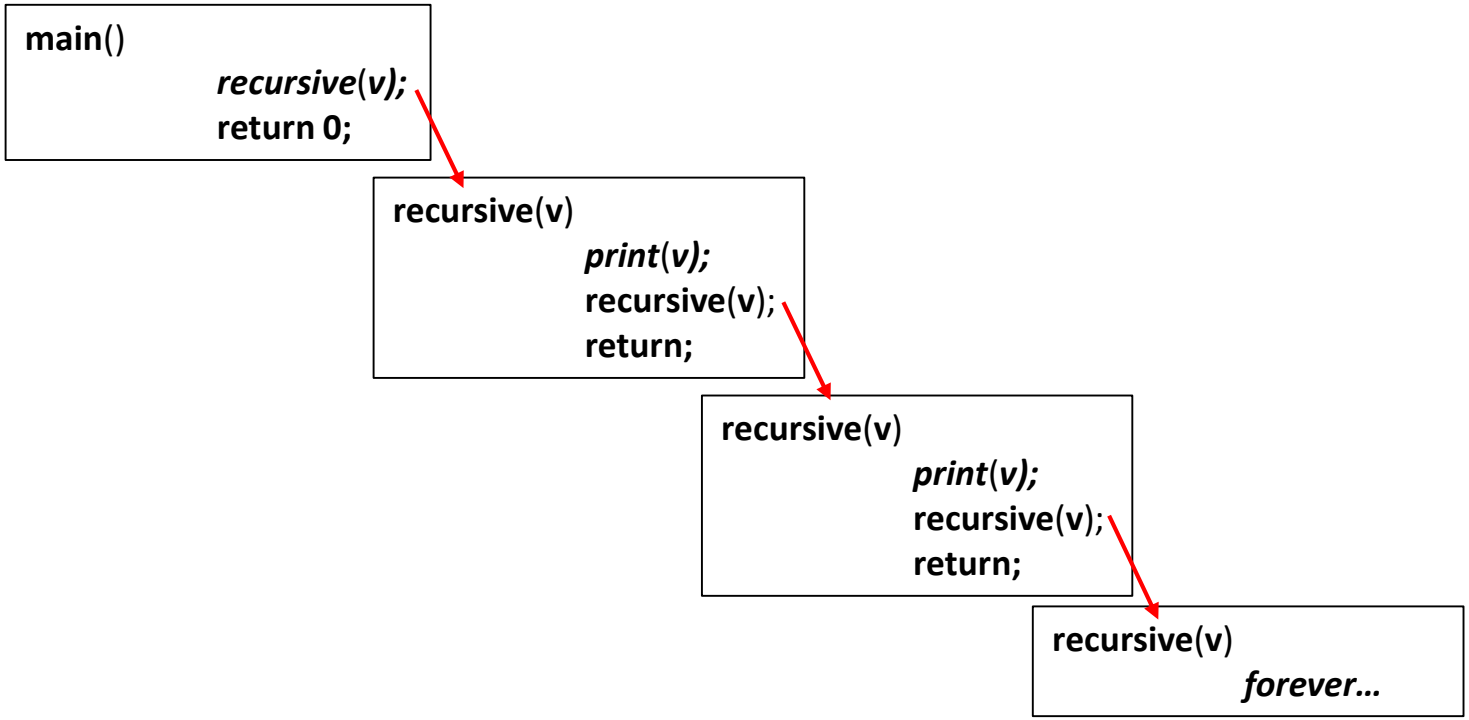




재귀 함수

재귀(Recursion)

- 자신을 정의할 때 자기 자신을 재참조하는 방법
  - 재귀 함수(Recursion Function)는 함수를 정의할 때 자기 자신이 포함됨



Graph 순회

그래프 순회 방법

- 깊이 우선 탐색 (Depth First Search)

  - 미로를 탐색할 때 갈 수 있을 때까지 계속 가다가 더 이상 갈 수 없으면 가까운 갈림길로 돌아와서 다시 탐색을 진행하는 방법과 유사
  - 넓게(wide)가 탐색하기 전에 깊게(deep) 탐색
  - BFS보다 구현이 간단 => 재귀함수를 이용

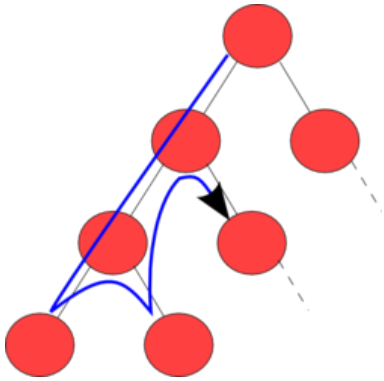
- 너비 우선 탐색 (Breadth First Search)

  - 깊게(deep) 탐색하기 전에 넓게(wide) 탐색
  - 두 정점 사이의 최단 경로 혹은 임의의 경로를 찾을때 이용

깊이 우선 탐색 (DFS)

깊이 우선 탐색 (DFS) 의 특징

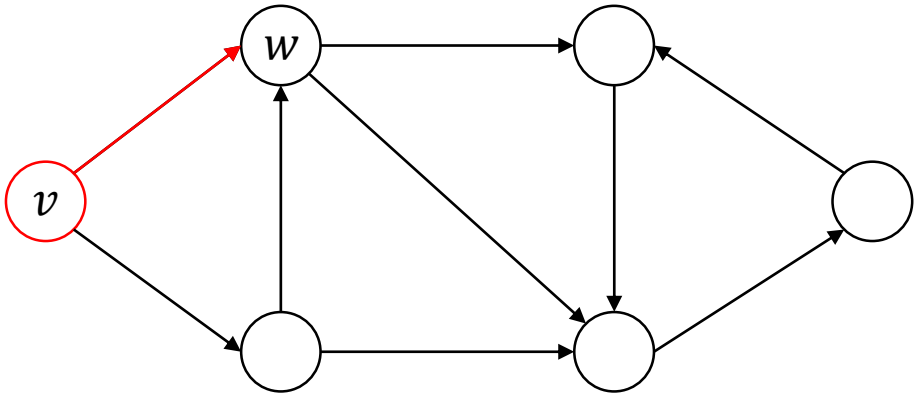
- 재귀 함수를 사용하여 간단하게 구현 가능
- 트리 순회도 Graph Traversal의 한 종류



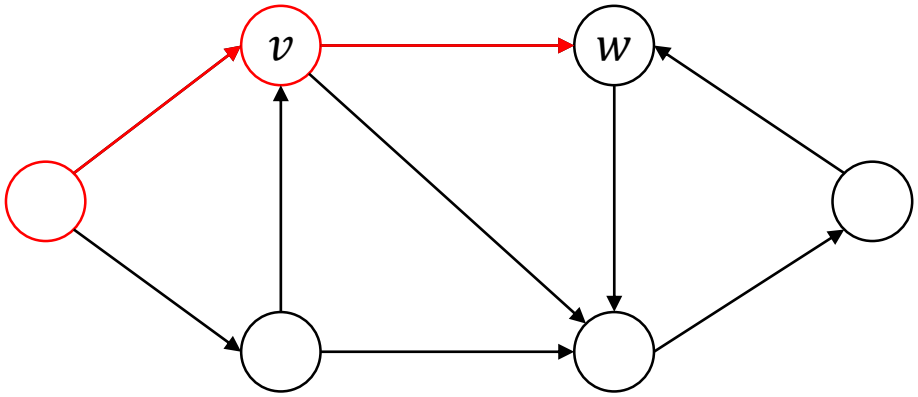
- 해당 정점을 방문 했었는지 여부를 반드시 기록해 야함

=> 하지 않으면, 무한 루프에 빠지게 된다.

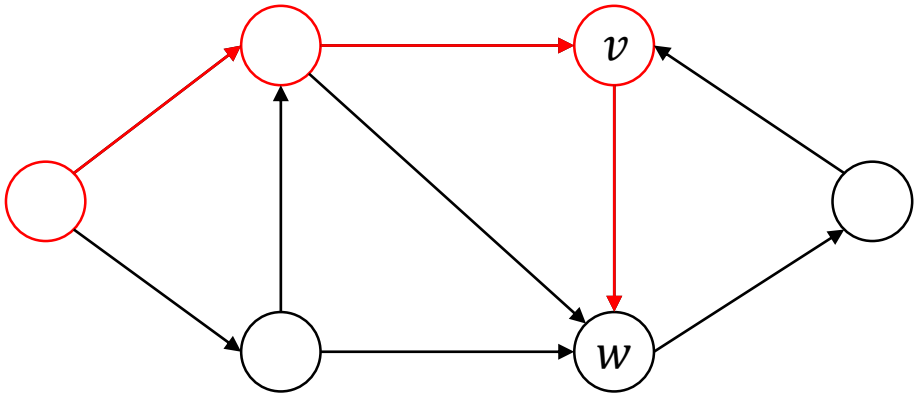
```
procedure DFS(G,v):
        label v as discovered
        for all edges from v to w in G.adjacentEdges(v) do
                if vertex w is not labeled as discovered then
                        recursively call DFS(G,w)
```



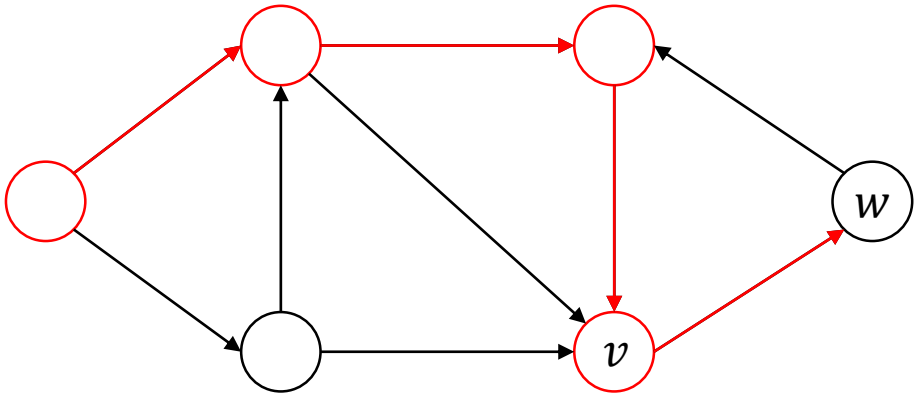
```
procedure DFS(G,v):
        label v as discovered
        for all edges from v to w in G.adjacentEdges(v) do
                if vertex w is not labeled as discovered then
                        recursively call DFS(G,w)
```



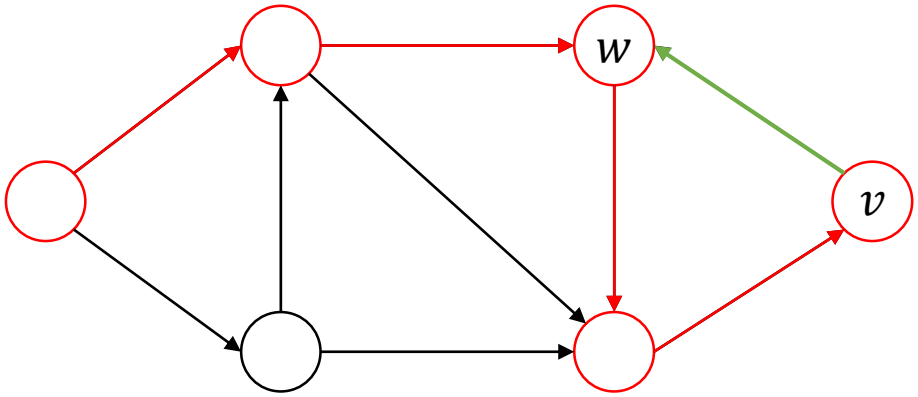
```
procedure DFS(G,v):
        label v as discovered
        for all edges from v to w in G.adjacentEdges(v) do
                if vertex w is not labeled as discovered then
                        recursively call DFS(G,w)
```



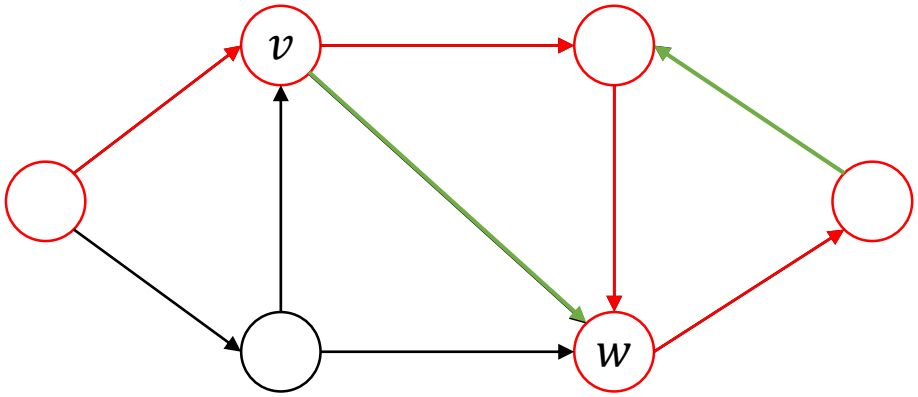
```
procedure DFS(G,v):
        label v as discovered
        for all edges from v to w in G.adjacentEdges(v) do
                if vertex w is not labeled as discovered then
                        recursively call DFS(G,w)
```



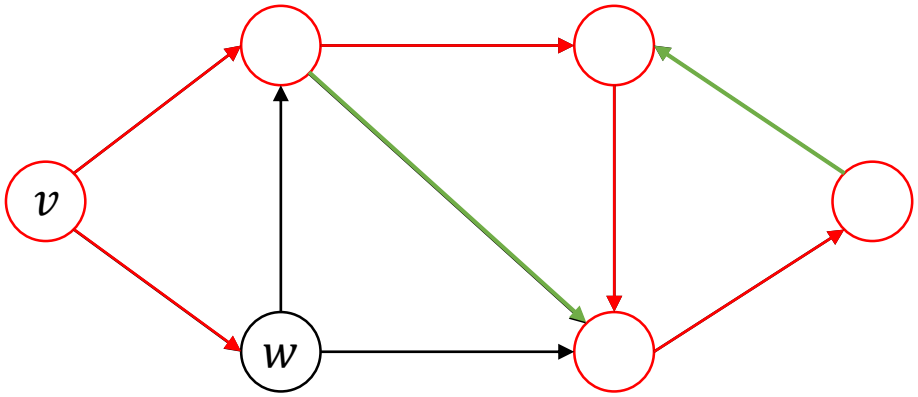
```
procedure DFS(G,v):
        label v as discovered
        for all edges from v to w in G.adjacentEdges(v) do
                if vertex w is not labeled as discovered then
                        recursively call DFS(G,w)
```



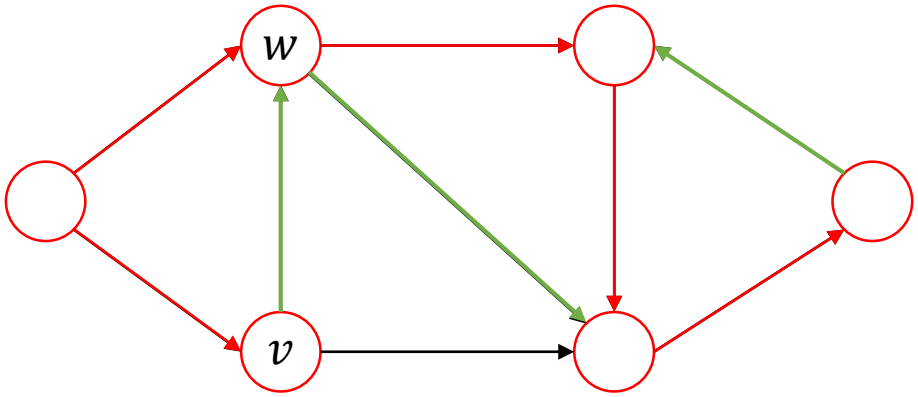
```
procedure DFS(G,v):
        label v as discovered
        for all edges from v to w in G.adjacentEdges(v) do
                if vertex w is not labeled as discovered then
                        recursively call DFS(G,w)
```



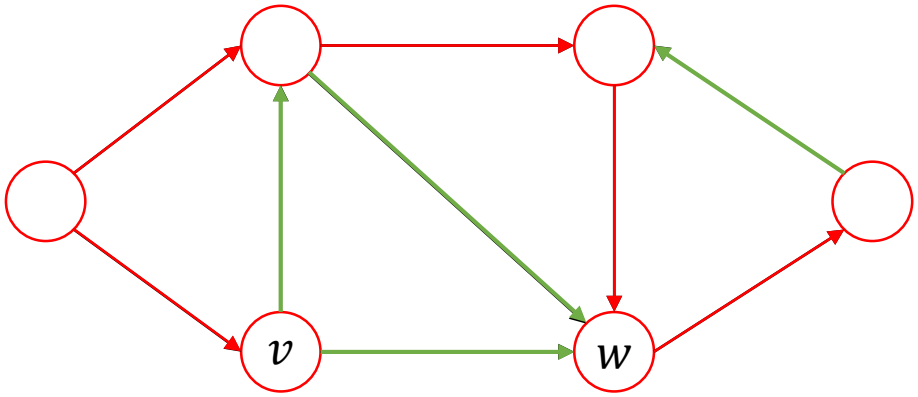
```
procedure DFS(G,v):
        label v as discovered
        for all edges from v to w in G.adjacentEdges(v) do
                if vertex w is not labeled as discovered then
                        recursively call DFS(G,w)
```



```
procedure DFS(G,v):
        label v as discovered
        for all edges from v to w in G.adjacentEdges(v) do
                if vertex w is not labeled as discovered then
                        recursively call DFS(G,w)
```



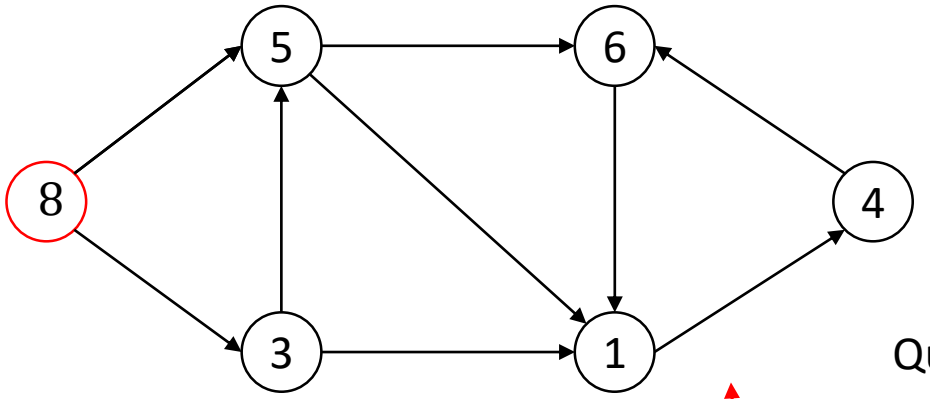
```
procedure DFS(G,v):
        label v as discovered
        for all edges from v to w in G.adjacentEdges(v) do
                if vertex w is not labeled as discovered then
                        recursively call DFS(G,w)
```



너비 우선 탐색 (BFS)

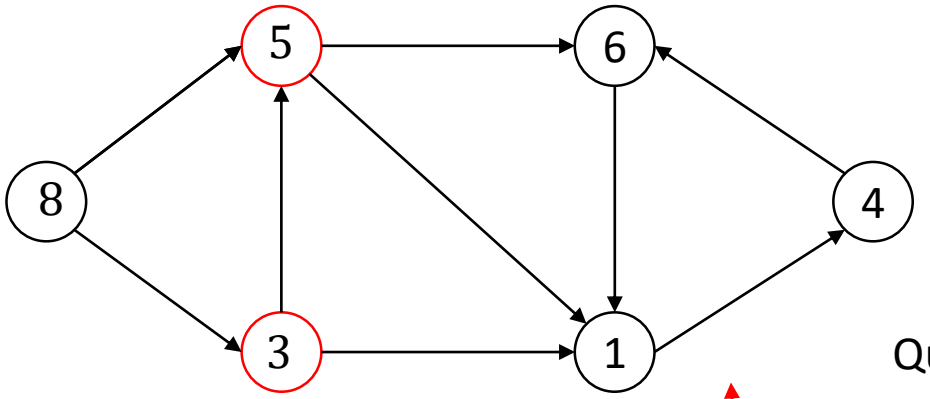
- 너비 우선 탐색 (BFS) 의 특징
  - Queue를 사용하여 간단하게 구현 가능
  - 거리가 가까운 정점부터 순차적으로 방문
  - 시작정점 s로부터 가장 짧은(간선들의 개수 기준) 경로의 길이에 따라 정점 분류 가능

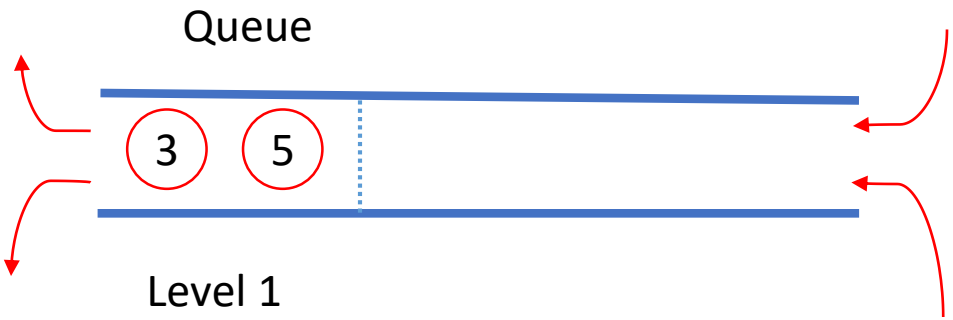
```
procedure BFS(G, s):
        Queue.push(s)
        while Queue is not empty
            v=Queue.pop()
            for each vertex v' adjacent on v
                if v' is unexplored
                    Queue.push(v')
                    label v' as discovered
```



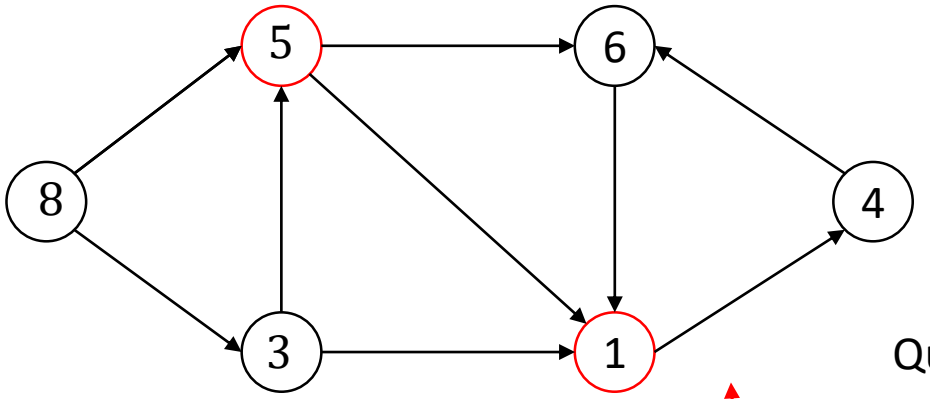
Queue

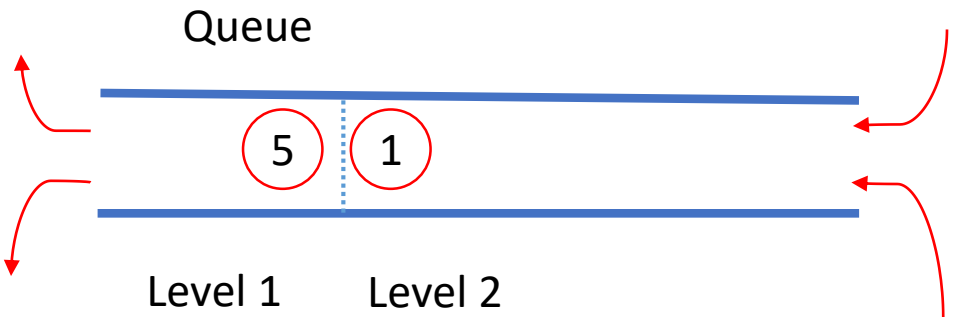
```
procedure BFS(G, s):
        Queue.push(s)
        while Queue is not empty
            v=Queue.pop()
            for each vertex v' adjacent on v
                if v' is unexplored
                    Queue.push(v')
                    label v' as discovered
```



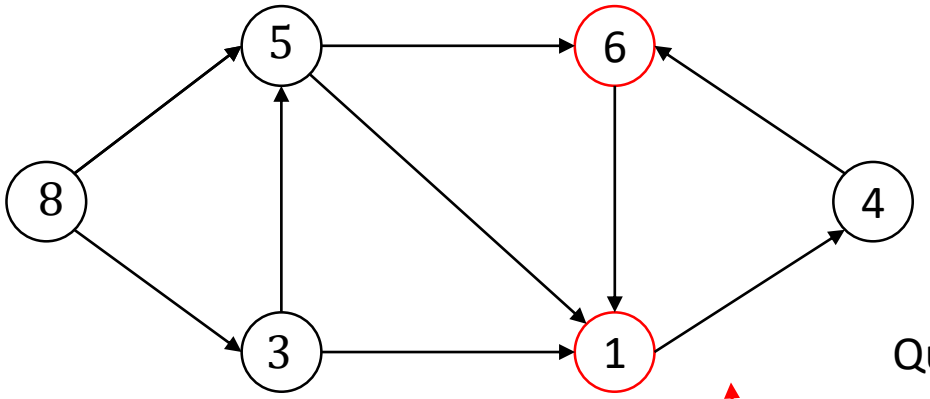


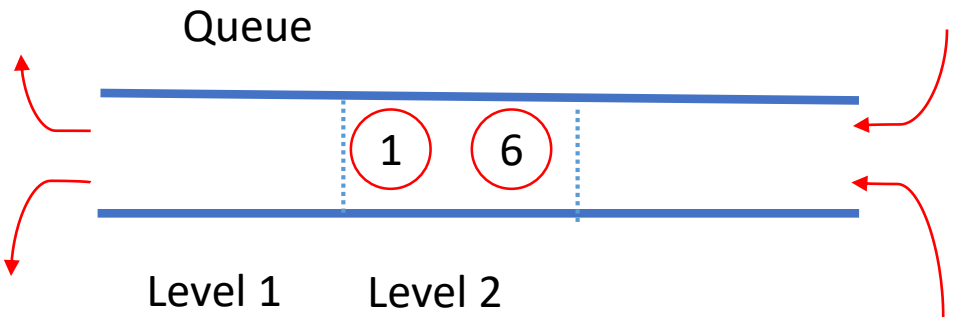
```
procedure BFS(G, s):
        Queue.push(s)
        while Queue is not empty
            v=Queue.pop()
            for each vertex v' adjacent on v
                if v' is unexplored
                    Queue.push(v')
                    label v' as discovered
```



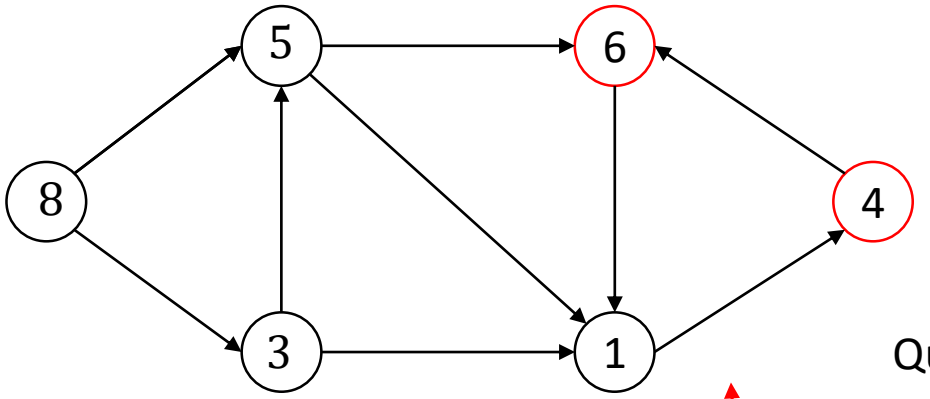


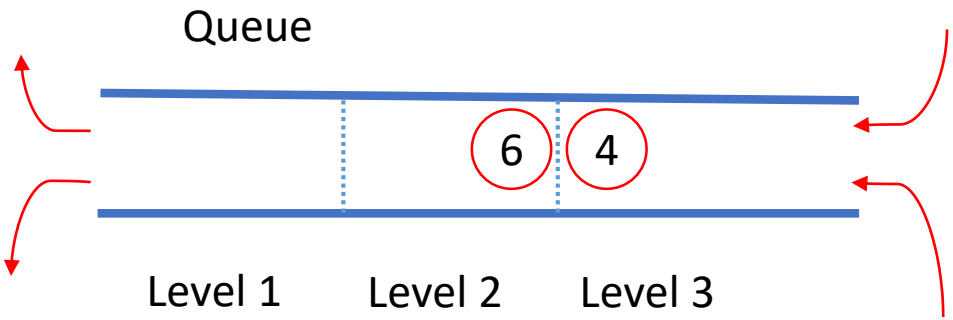
```
procedure BFS(G, s):
        Queue.push(s)
        while Queue is not empty
            v=Queue.pop()
            for each vertex v' adjacent on v
                if v' is unexplored
                    Queue.push(v')
                    label v' as discovered
```



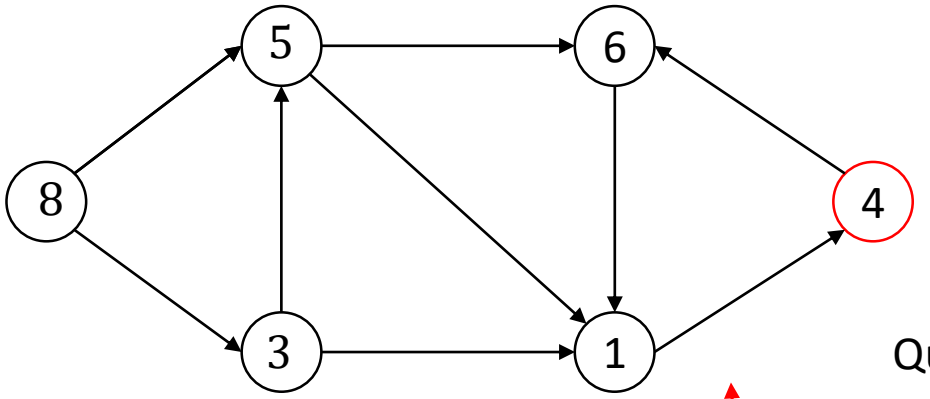


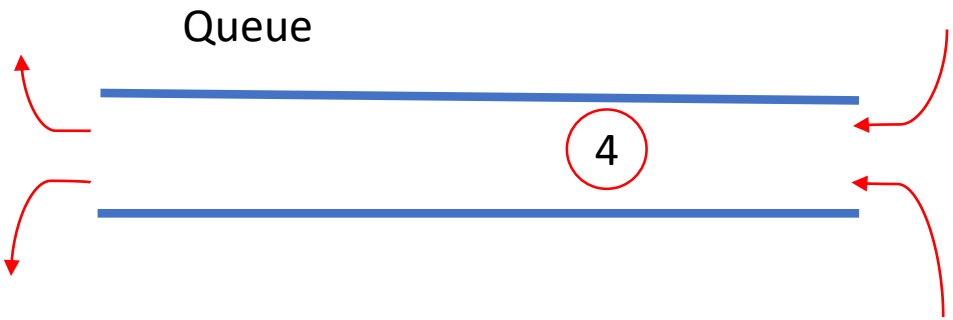
```
procedure BFS(G, s):
        Queue.push(s)
        while Queue is not empty
            v=Queue.pop()
            for each vertex v' adjacent on v
                if v' is unexplored
                    Queue.push(v')
                    label v' as discovered
```



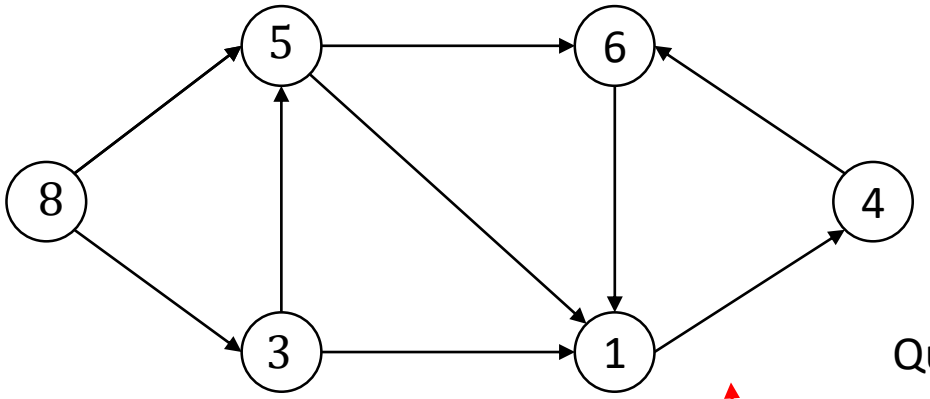


```
procedure BFS(G, s):
        Queue.push(s)
        while Queue is not empty
            v=Queue.pop()
            for each vertex v' adjacent on v
                if v' is unexplored
                    Queue.push(v')
                    label v' as discovered
```





```
procedure BFS(G, s):
        Queue.push(s)
        while Queue is not empty
            v=Queue.pop()
            for each vertex v' adjacent on v
                if v' is unexplored
                    Queue.push(v')
                    label v' as discovered
```



Queue

코드(Adjacency Vertex List 기반)

```cpp
#include <iostream>
#include <vector>
#define NOT_EXPLORED 0
#define DISCOVERY 1
#define BACK 2

using namespace std;

class Vertex {
public:
    int data;
    int degree;
    bool visited; // check vertex is visited or not by DFS
    vector<Vertex*> adj_list;

    Vertex(int data) {
        this->data = data;
        this->degree = 0;
        this->visited = false;
    }
};

class Edge {
public:
    Vertex* src;
    Vertex* dst;
    int data;
    int edge_stat; // edge status

    Edge(Vertex* src, Vertex* dst) {
        this->src = src;
        this->dst = dst;
        this->edge_stat = NOT_EXPLORED;
    }
};

```

코드(Adjacency Vertex List 기반)


```
37  class Graph {
38  private:
39      vector<Vertex*> vertex_list;
40      vector<Edge*> edge_list;
41  public:
42      Graph( ) {
43      }
44
45      Vertex* findvertex(int data) {
46          Vertex* v = NULL;
47          for (int i = 0; i < vertex_list.size(); i++) {
48              if (vertex_list[i]->data == data) {
49                  v = vertex_list[i];
50                  break;
51              }
52          }
53          return v;
54      }
55
56      Edge* findedge(Vertex* src, Vertex* dst) {
57          Edge* e = NULL;
58          for (int i = 0; i < edge_list.size(); i++) {
59              if (edge_list[i]->src == src && edge_list[i]->dst == dst) {
60                  e = edge_list[i];
61                  break;
62              }
63              else if (edge_list[i]->src == dst && edge_list[i]->dst == src) {
64                  e = edge_list[i];
65                  break;
66              }
67          }
68          return e;
69      } // adjacency matrix를 이용할 경우, 상수타임에 체크 가능
70
71      void insert_vertex(int data) {
72          if (findvertex(data) == NULL) {
73              Vertex* new_v = new Vertex(data);
74              vertex_list.push_back(new_v);
75          }
76      }
```

코드(Adjacency Vertex List 기반)

```
71      void insert_vertex(int data) {
72          if (findvertex(data) == NULL) {
73              Vertex* new_v = new Vertex(data);
74              vertex_list.push_back(new_v);
75          }
76      }
77
78      void insert_edge(int src_data, int dst_data) {
79          Vertex* src = findvertex(src_data);
80          Vertex* dst = findvertex(dst_data);
81          if (findedge(src, dst) == NULL) {
82              Edge* new_e = new Edge(src, dst);    // generate edge
83              edge_list.push_back(new_e);
84              src->adj_list.push_back(dst);          // insert vertex to their adj list
85              dst->adj_list.push_back(src);
86          }
87          else cout << -1 << endl;
88          src->degree++;
89          dst->degree++;
90      }
91
92      void dfs(Vertex* v) { ... }
109 };
110
```

코드(Adjacency Edge List 기반)

```
#include<iostream>
#include<string>
#include<vector>
using namespace std;

class DoublyEdgeLinkedList;

class vertex {
public:
    DoublyEdgeLinkedList *incidentEdgeList;
    int degree = 0;
    int data;
    bool visited = false;
    vertex *prev;
    vertex *next;
    vertex(int data);
    void increase_degree() {
        this->degree++;
    }
    void decrease_degree() {
        this->degree--;
    }
};
```

코드(Adjacency Edge List 기반)

```
class edge {
public:
    edge* prev;
    edge* next;
    edge* myselfInFisrtincidentEdge;
    edge* myselfInSecondincidentEdge;
    edge* myselfInTotalEdgeList;
    vertex* source;
    vertex* destination;
    string word;
    bool explore;
    bool discovery;
    bool back;
    edge(vertex* a, vertex* b, string word) {
        this->source = a;
        this->destination = b;
        this->myselfInFisrtincidentEdge = NULL;
        this->myselfInSecondincidentEdge = NULL;
        this->myselfInTotalEdgeList = NULL;
        this->word = word;
        this->explore = false;
        this->discovery = false;
        this->back = false;
    }
};
```

코드(Adjacency Edge List 기반)

```
class DoublyEdgeLinkedList {
public:
    edge *head;
    edge *tail;
    int size;
    DoublyEdgeLinkedList() {
        this->head = NULL;
        this->tail = NULL;
        this->size = 0;
    }
    void insert(edge *insertEdge) {
        if (this->head == NULL) {
            head = insertEdge;
            tail = insertEdge;
        }
        else {
            tail->next = insertEdge;
            insertEdge->prev = tail;
            tail = insertEdge;
        }
        this->size++;
    }
    void remove(edge *delEdge) {

        if (delEdge == head || delEdge == tail) {
            if (delEdge == head && delEdge != tail) {
                edge *temp = head;
                head = head->next;
                head->prev = NULL;
                delete temp;

            }
            else if (delEdge == tail && delEdge != head) {
                edge *temp = tail;
                tail = tail->prev;
                tail->next = NULL;
                delete temp;

            }
            else { head = tail = NULL; }
        }
        else {
            delEdge->prev->next = delEdge->next;
            delEdge->next->prev = delEdge->prev;
            delete delEdge;
        }
        this->size--;
    }
};
```

코드(Adjacency Edge List 기반)


```
vertex::vertex(int data) {
    this->degree = 0;
    this->data = data;
    this->incidentEdgeList = new DoublyEdgeLinkedList();
}

class DoublyVertexLinkedList {
public:
    vertex *head;
    vertex *tail;
    int size;
    DoublyVertexLinkedList() {
        this->head = NULL;
        this->tail = NULL;
        this->size = 0;
    }
    void insert(vertex *insertVertex) {
        if (this->head == NULL) {
            head = insertVertex;
            tail = insertVertex;
        }
        else {
            tail->next = insertVertex;
            insertVertex->prev = tail;
            tail = insertVertex;
        }
        this->size++;
    }
    void remove(vertex *delVertex) {
        if (delVertex == head || delVertex == tail) {
            if (delVertex == head && delVertex != tail) {
                vertex *temp = head;
                head = head->next;
                head->prev = NULL;
                delete temp;
            }
            else if (delVertex == tail && delVertex != head) {
                vertex *temp = tail;
                tail = tail->prev;
                tail->next = NULL;
                delete temp;
            }
            else { head = tail = NULL; }
        }
        else {
            delVertex->prev->next = delVertex->next;
            delVertex->next->prev = delVertex->prev;
            delete delVertex;
        }
        this->size--;
```

코드(Adjacency Edge List 기반)


```
class graph {
public:
    DoublyVertexLinkedList* TotalvertexList;
    DoublyEdgeLinkedList* TotaledgeList;
    int vertexSize;
    int maxSize;
    graph() {
        this->vertexSize = 0;
        this->TotalvertexList = new DoublyVertexLinkedList();
        this->TotaledgeList = new DoublyEdgeLinkedList();
    }
    bool isFindVertex(int data) {
        vertex *tempVertex;
        bool flag = false;
        tempVertex = TotalvertexList->head;
        while (tempVertex != NULL) {
            if (tempVertex->data == data)
            {
                flag = true; break;
            }
            tempVertex = tempVertex->next;
        }
        return flag;
    }
    vertex* findVertex(int data) {
        vertex *tempVertex;
        tempVertex = TotalvertexList->head;
        while (tempVertex != NULL) {
            if (tempVertex->data == data)
            {
                break;
            }
            tempVertex = tempVertex->next;
        }
```

코드(Adjacency Edge List 기반)

```
bool isFindEdge(int source, int destination) {
    edge* tempEdge;
    bool flag = false;
    tempEdge = TotaledgeList->head;
    while (tempEdge != NULL) {
        if (tempEdge->source->data == source &&tempEdge->destination->data == destination ||
            tempEdge->source->data == destination &&tempEdge->destination->data == source)
        {
            flag = true; break;
        }
        tempEdge = tempEdge->next;
    }
    return flag;
}

edge* findEdge(int source, int destination) {
    edge* tempEdge;
    tempEdge = TotaledgeList->head;
    while (tempEdge != NULL) {
        if (tempEdge->source->data == source &&tempEdge->destination->data == destination ||
            tempEdge->source->data == destination &&tempEdge->destination->data == source)
        {
            break;
        }
        tempEdge = tempEdge->next;
    }
    return tempEdge;
}
```

코드(Adjacency Edge List 기반)

```
void insert_vertex(int n) {
    if (isFindVertex(n) == true)return;
    else {
        vertex* newVertex = new vertex(n);
        TotalvertexList->insert(newVertex);
        this->vertexSize++;
    }
}
void insert_edge(int source, int destination,string word) {
    if (isFindVertex(source) == true && isFindVertex(destination) == true) {
        vertex* srcVertex = findVertex(source);
        vertex* dstVertex = findVertex(destination);
        edge* newEdge = new edge(srcVertex, dstVertex,word);        //totaledgelist에 추가될 newedge

        TotaledgeList->insert(newEdge);

        edge* tempEdge1 = new edge(srcVertex, dstVertex, word);     //src.incidentedges 에 추가될 new edge
        edge* tempEdge2 = new edge(srcVertex, dstVertex, word);
        tempEdge1->myselfInTotalEdgeList = newEdge;
        tempEdge2->myselfInTotalEdgeList = newEdge;

        srcVertex->incidentEdgeList->insert(tempEdge1);
        dstVertex->incidentEdgeList->insert(tempEdge2);
        newEdge->myselfInFisrtincidentEdge = tempEdge1;
        newEdge->myselfInSecondincidentEdge = tempEdge2;

        srcVertex->increase_degree();
        dstVertex->increase_degree();
    }
    else return;
}
```

코드(Adjacency Edge List 기반)

```
    void DFS(vertex *curV) {

        ////////////////////

        //직접 코딩하세요 //////

        ////////////////////

};
```


Thank you